GPU Parallelization of Algebraic Dynamic Programming

Peter Steffen, Robert Giegerich, Mathieu Giraud

GPU Parallelization of Algebraic Dynamic Programming

Peter Steffen[1,2], Robert Giegerich[1], and Mathieu Giraud[2]

[1] Technische Fakultät, Universität Bielefeld, D-33501 Bielefeld, Germany
psteffen@techfak.uni-bielefeld.de, robert@techfak.uni-bielefeld.de
[2] CNRS, LIFL, Université Lille 1, 59 655 Villeneuve d'Ascq cedex, France
mathieu.giraud@lifl.fr

Abstract. Algebraic Dynamic Programming (ADP) is a framework to encode a broad range of optimization problems, including common bioinformatics problems like RNA folding or pairwise sequence alignment. The ADP compiler translates such ADP programs into C. As all the ADP problems have similar data dependencies in the dynamic programming tables, a generic parallelization is possible. We updated the compiler to include a parallel backend, launching a large number of independent threads. Depending on the application, we report speedups ranging from 6.1× to 25.8× on a Nvidia GTX 280 through the CUDA libraries.

1 Introduction

Dynamic programming in bioinformatics. In biological sequence analysis, there arise numerous combinatorial optimization problems that are solved by dynamic programming. Pattern matching in DNA or protein sequences, comparison for local or global similarity, and structure prediction from RNA sequences are frequent tasks, as well as the modeling of families of proteins and RNA structures with the widely used Hidden Markov Models (HMMs) and stochastic context free grammars (SCFG), respectively [5]. The scoring schemes associated with these optimization problems can be quite sophisticated. The thermodynamic model for RNA structure prediction, for example, has more than thousand parameters. This requires elaborate case analysis. Objective functions often ask for more than a single answer, such as the best non-overlapping pattern hits to a genome above a certain score threshold. Finally, biological sequences tend to be long (from 77 characters for a tRNA, 10000 for a gene, $3 * 10^6$ for a bacterial genome, to the $3 * 10^9$ nucleotides of a mammalian genome such as human or mouse). The time and space requirements for a dynamic programming algorithm are often limiting factors for the problems the biologists need to solve. The development of reliable and efficient dynamic programming algorithms in bioinformatics is a recurring challenge, in sharp contrast to the simplicity suggested by the textbook examples of dynamic programming which we use to teach computer science students.

Algebraic dynamic programming. In all these optimization problems, the logical problem decomposition follows the decomposition of the input sequence

into subwords. It has been observed early that the resulting dynamic programming recurrences strongly resemble those of a Cocke-Younger-Kasami [3] type parsing algorithm [20]. Pursuing this analogy, we have developed an algebraic style of dynamic programming (ADP) over sequential data. The search space of the optimization problem at hand is described by a *yield grammar*, which is a regular tree grammar generating a tree language, and implicitly a context-free language as the set of leaf sequences of these trees. Scoring and optimization are described by an *evaluation algebra*, which interprets the tree operators as functions that compute local score contributions, and hence solve larger problems when given optimal solutions of smaller ones, consistent with the general paradigm of dynamic programming. This leads to a complete specification of dynamic programming algorithms on a rather high level of abstraction.

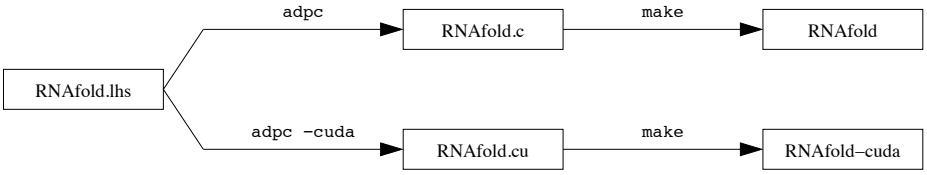

Fig. 1. ADP workflow. The goal of this study is to conceive and implement an automatic GPU parallelization (bottom).

General-purpose computation on GPU. For a few years, issues with heat dissipation have prevented the processors from having higher frequencies. One of the answers to maintain the Moore's Law is the use of parallel processing with massively manycore architectures. Graphic processing units (GPUs) are a first step towards those architectures, and recent trends blur the line between such GPUs and multi-core processors.

GPUs were used in bioinformatics since 2005 for phylogenetic studies [4], then for multiple sequence alignment based on an optimized Smith-Waterman implementation [10]. The CUDA libraries, first released in 2007 [2], have deeply simplified the development on GPUs. Recent papers provide speedups on applications involving suffix trees [19] or again Smith-Waterman comparisons [9, 11, 13], error correction in DNA short-reads sequencing [21], computation with position weight matrices [8], RNA folding [18], and neighbor-joining trees for multiple sequence alignments [12].

The current Nvidia architectures [2] offer two levels of parallelism. For the coarse-grained level, several multiprocessors execute *blocks* of independent computations. Each multiprocessor is then a kind of large SIMD device, able to process several different fine-grained *threads* at a given time. All those threads are executing exactly the same instructions: if a *divergence* in a condition occurs, the branches of the condition are serialized.

Contents. In this paper we describe an approach to extend the ADP compiler such that it generates parallel programs (Figure 1). This approach has been implemented in the ADP compiler with the CUDA libraries [2]. Our new contribution is thus a generic method to create parallel CUDA programs for bioinformatics applications – classical and yet-to-be written ones. The next section presents background on Algebraic Dynamic Programming. Section 3 presents the GPU parallelization of ADP. Section 4 gives some results and discussion: depending on the application, we get speedups ranging from 6.1× to 25.8× on a Nvidia GTX 280.

2 Algebraic Dynamic Programming

We briefly introduce the Algebraic dynamic programming (ADP) methodology on a simple Nussinov type RNA secondary structure prediction problem, the maximization of the number of base pairs [15]. Section 4 reports results for several other applications. See [6] for a complete presentation of the ADP method.

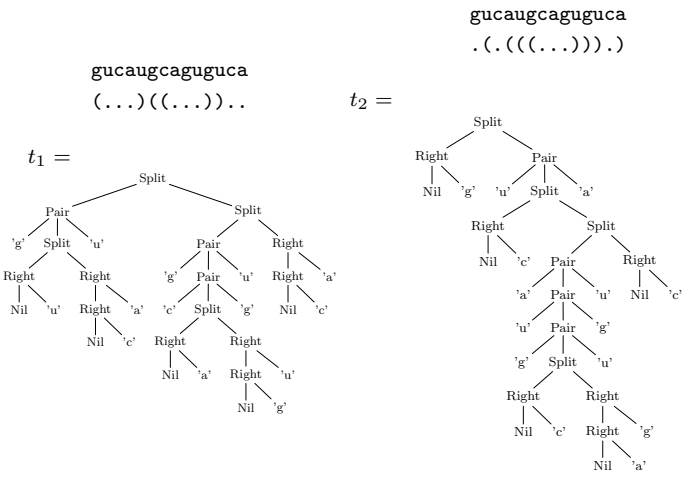

Fig. 2. Two candidates in the search space for the best secondary structure for the sequence `gucaugcaguguca`.

When designing a dynamic programming algorithm in algebraic style, we need to specify four constituents: the input alphabet, the search space, the scoring of the candidates, and the objective function.

Alphabet. The input RNA sequence is a string over the alphabet $\mathcal{A} = \{a, c, g, u\}$.

Search space. Given an input sequence $w \in \mathcal{A}^*$, the search space is the set of all possible secondary structures the sequence w can form. In the ADP terminology,

the elements of the search space for a given input sequence are called *candidates*. Figure 2 gives example of candidates for RNA folding. This tree representation of candidates exists for any application of dynamic programming [23]. To describe the candidates, the ADP methodology uses the notion of *tree grammar*. Figure 3 shows the grammar `nussinov78`, origin of our two example trees. For each sequence $w \in \mathcal{A}^*$, the grammar defines a search space $P_{\mathcal{G}}(w)$ that is the set of all parses of the sequence w for $\mathcal{G}$.

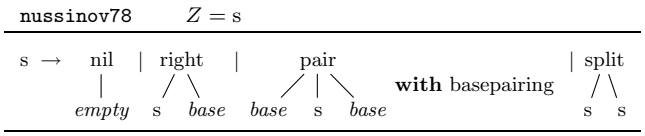

Fig. 3. Tree grammar `nussinov78` consists of one production with four alternatives. Symbol Z denotes the axiom of the grammar.

Scoring and objective. Given an element of the search space as a tree $t \in P_{\mathcal{G}}(w)$, we need to score this element. In our example we are only interested in counting base pairs, so scoring is very simple: the score of a tree is the number of *pair*-nodes in t. For the two candidates of Figure 2 we obtain scores of 3 (t_1) and 4 (t_2). Moreover, we need to choose one or several solutions from the pool of candidates. For this purpose we add an objective function h which chooses one or more elements from a list of candidate scores.

```
bpmax = (nil, right, pair, split, h) where
nil(s)        = 0
right(s,b)    = s
pair(a,s,b)   = s + 1
split(s,s')   = s + s'
h([])         = []
h([s_1,...,s_r]) = [max_{1≤i≤r} s_i]
```

Scoring schemes with objective functions are called *evaluation algebras* in ADP. The above example is the evaluation algebra `bpmax` for maximizing the number of base pairs. The flexibility of the algebraic approach lies in the fact that we don't have to stop with definition of *one* algebra: simply define another algebra and get other results for the same search space. We use the notation $\mathcal{E}(t)$ to indicate the value obtained from t under evaluation with algebra $\mathcal{E}$. All that is left to do is to evaluate the candidates in a given algebra, and make our choice via the objective function h. For example, candidates t_1 and t_2 of Figure 2 are evaluated by algebra `bpmax`, with $h(\texttt{bpmax}(t_1), \texttt{bpmax}(t_2)) = [\max(3,4)] = [4]$.

This example was fairly simple: complete RNA folding algorithms are typically based on energy minimization, and include energies of stacking regions (or helices), bulge loops, internal loops, hairpin loops and multiple loops. Figure 4

shows an excerpt of the real `RNAfold.lhs` grammar that includes the full Turner model [14]. The grammar can be read as a standard context-free grammar. The operator `~~~` connects succeeding symbols and the operator `|||` divides alternative productions for a non-terminal. The symbol `<<<` denotes application of an algebra function and `...` denotes application of the evaluation function `h`. Finally, the operator `with` denotes the use of a filter function, that means that `(base ~~~ closed ~~~ base) ‘with‘ basepairing` is only successful if the two bases can form a base pair.

```
rnafold alg f = axiom struct where
 (sadd,cadd,is,sr,hl,bl,br, il, il11, il12, il21, il22,
  dl, dr, dlr, edl, edr, edlr, drem, cons, ul, pul, addss, ssadd, nil, combine, h) = alg

  struct        = tabulated (sadd <<< base     ~~~ struct |||
                             cadd <<< initstem ~~~ struct |||
                             nil  <<< empty ... h)

  initstem = tabulated (is <<< loc ~~~ closed ~~~ loc ... h)
  closed   = tabulated (stack ||| ((hairpin ||| leftB ||| rightB ||| iloop ||| multiloop)
                  ‘with‘ stackpairing) ... h)

  stack     = (sr <<< base ~~~ closed ~~~ base)  ‘with‘ basepairing ... h
  hairpin   = hl <<< base ~~~ base ~~~ (region ‘with‘ (minsize 3))   ~~~ base ~~~ base ... h
  leftB     = bl <<< base ~~~ base ~~~ region   ~~~ initstem   ~~~ base ~~~ base ... h
  rightB    = br <<< base ~~~ base ~~~ initstem ~~~ region     ~~~ base ~~~ base ... h
  iloop     = il <<< base ~~~ base ~~~ (region  ‘with‘ (maxsize 30)) ~~~ closed ~~~
                                       (region  ‘with‘ (maxsize 30)) ~~~ base ~~~ base ... h
```

Fig. 4. Excerpt from the ADP grammar `RNAfold.lhs`. The complete grammar includes further productions for multiloop structures.

3 Automatic Parallelization of ADP

Principle. A compiler that translates ADP programs into C was previously developed [7]. This task includes some advanced optimization techniques, see [22] for a detailed overview. With the option `-cuda`, the compiler is now switched into the CUDA code generation mode. The compiler uses the same backend both for CPU and GPU code generation and only differs in the following parts:

1. The dynamic programming tables need to be stored both on the host and on the global memory of the GPU. The compiler generates code to synchronize these tables.

2. For each dynamic programming table, the compiler generates a calculation function. This is the same function both for CPU and GPU mode, so the only change is that it is declared to be executed as a GPU kernel. Figure 5 shows the CUDA code for the dynamic programming main loop in the RNA secondary structure prediction program. The kernel function, `calc_all`, contains the calls for the calculation of the six dynamic programming tables.

3. In CPU mode, all table elements are calculated sequentially with increasing subword length. This order of computation has to be changed to enable parallelization. For the RNA secondary structure prediction program, the calculation of a table element (i, j) depends only on the elements in the triangle in the lower left (see Figure 6, on the left). So all elements in one diagonal can be calculated in parallel. The whole dynamic programming table is then calculated in a loop over all diagonals (see Figure 5). This approach can be generalized to all dynamic programming algorithms over sequence data: in all ADP grammars, all results are combined from results of shorter subsequences. Therefore, the calculation of a table element (i, j) depends only on results that lie between the indices (i, j).

All these changes are done automatically by the compiler and do not require any changes to the ADP grammar. The number of blocks and threads used for the calculation can be configured as a parameter at runtime.

```
__global__ static void calc_all(int diag, int n) {
   int i = blockIdx.x*blockDim.x+threadIdx.x;
   int j = i + diag;
   if ((i <= n) && (j <= n)) {
      calc_closed(i, j);
      calc_initstem(i, j);
      calc_struct(i, j);
      calc_block(i, j);
      calc_comps(i, j);
   }
}

static void mainloop() {
   for (int diag=0; diag<=n; diag++) {
      (...)
      calc_all <<< grid, threads >>> (diag, n);
   }
}
```

Fig. 5. Kernel and main CUDA code for the dynamic programming main loop corresponding to the grammar shown on Figure 4. Note that each kernel thread also computes inner loops for the folding calculations. Actual codes are available on the ADP website (`http://bibiserv.techfak.uni-bielefeld.de/adp/cuda.html`).

Window mode. It does not make any sense to fold a complete genome as a single RNA molecule. This remark is the same for other applications: for example, a thermodynamic matcher [17] looks for some small sub-sequences (50 to some

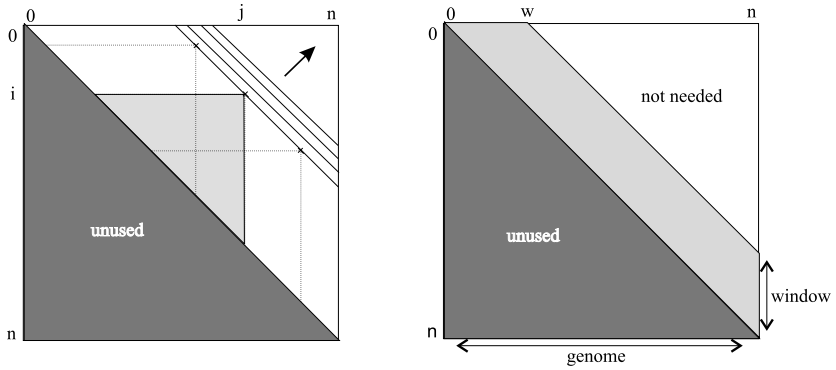

Fig. 6. Left: data dependencies for RNA secondary structure prediction. The computation of the table element (i, j) needs the $O((j-i)^2)$ elements in the underlying triangle. Right: window mode. With a large genome (of size n), we just need to fold sequences on small windows (of size w).

hundreds bases) matching a given structural pattern in a large sequence. In those applications, we need to compute only some diagonals above the main diagonal (Figure 6, on the right). The option `-cudaw` sets the ADP compiler in window mode. Whereas the CPU version sequentially computes all windows, the GPU version loads a large sequence into the GPU and launches a large number of independent threads, thus increasing the parallelism. This is also done automatically by the compiler and does not require any changes to the source program.

4 Results

Table 1 shows the results on three different applications with RNA sequences: *RNAfold* (see previous section), *pknotsRG* (detection of pseudo-knots [16]), and a tRNA thermodynamic matcher. The program *pknotsRG* predicts RNA secondary structures including a restricted class of pseudoknots. The thermodynamic matcher was created by the graphical tool *Locomotif* [17]. Practical 9.9×, 14.5× and 6.1× speedups are obtained on those real applications with a GTX 280. In these speedups, the main bottlenecks are in memory transfers, as only the global memory of the GPU is used.

In the original `RNAfold.lhs` grammar, a part of production `calc_closed` is in fact computed for only 6 out of the 16 possible basepairs (filter `'with' stackpairing` on Figure 4). This brings a large divergence between the threads and breaks the GPU SIMD model. To confirm this fact, we tested a special version of this grammar, `RNAfold-bp.lhs`, that computes for every basepair the full recurrence equations (penalizing non-pairs): the speedup with the GTX 280 is almost doubled. This indicates that a similar speedup would result for the calculation of stochastic grammars, since here arbitrary base pairs are considered.

Grammar, window size, time complexity			PC1 Core2 + GeForce 8800			PC2 Xeon + GTX 280		
			CPU	GPU	speedup	CPU	GPU	speedup
RNAfold-bp.lhs	`-w 80`	$O(w^2n)$	176.09	19.22	9.1×	133.77	5.18	25.8×
RNAfold.lhs	`-w 80`	$O(w^2n)$	43.43	8.08	5.4×	35.57	3.59	9.9×
tRNA-matcher.lhs	`-w 100`	$O(w^2n)$	52.46	6.76	7.8×	43.60	3.01	14.5×
pknotRG.lhs	`-w 80`	$O(w^3n)$	26.82	10.64	2.5×	23.54	3.25	7.2×
pknotRG.lhs	`-w 160`	$O(w^3n)$	188.68	87.65	2.2×	166.27	27.22	6.1×

Table 1. Time (real times, in seconds) for executing different ADP grammars. CPU versions are compiled with `adpc`, and executed on a 2.4 GHz Core2 processor (PC1, 1 core used). and on a 3.0 GHz Xeon X5472 processor (PC2, 1 core used). GPU CUDA versions are compiled `adpc -cudaw`, and executed on a GeForce 8800 (PC1) and on a GTX 280 (PC2). Because of its increased number of cores and of its better handling of uncoalesced memory loads, the GTX 280 gives better speedups than the GeForce 8880. Note that the performance of the CPU does not impact the times reported for the GPU versions. For example, for `RNAfold`, the 19.22s for the PC1 GPU include only 0.20s of non-kernel computations, mainly for traceback in the DP matrix. Tests on `RNAfold` and `tRNA-matcher` were done on the 160 kbp genome of Candidatus Carsonella ruddii (Genbank reference `NC_008512`). Tests on `pknotsRG` were done on the first 20 kbp of the same genome.

It should be noted that our speedups are lower than the best possible ones. For example, Rizk and Lavenier [18] developed an optimized GPU RNAfold implementation: in particular, they pack together the 6/16 computations corresponding to the production `calc_closed`. They obtain a 17× speedup on a GTX 280 against one core of a 2.66 GHz Xeon (applied on a whole sequence, without window mode), whereas our best speedup without window mode is only 2.8× (results not shown). However, as our approach is generic, it can be applied on several algorithms with few efforts to the user.

On *pknotsRG*, runs with $w = 160$ get a little smaller speedup than with $w = 80$. As w is fixed, this does not limit the scalability of our approach: the input data size, n, can always grow with the same speedup.

Current limitations. Whereas grammars involving several sequences can be encoded in the ADP formalism, the ADP compiler now only works for one input sequence. Removing this limitation would allow to study other dynamic programming problems, as for example Smith-Waterman sequence alignment or RNA co-folding [6]. Finally, for some grammars (including the tRNA matcher), the ADP automatic table design generates some recursive functions, and those functions cannot be compiled with the CUDA libraries (there is no stack on the current cards). This automatic table design is removed through omitting the `-cto` option, but, in this case, the grammar should specify precisely which symbols of the grammar are to be "tabulated".

5 Perspectives

We implemented a parallel GPU CUDA backend for the ADP compiler, which works out-of-the-box for several grammars dealing with RNA sequences. The new ADP compiler and some example codes are available on the ADP website (`http://bibiserv.techfak.uni-bielefeld.de/adp/cuda.html`). Our approach is generic and requires few efforts to the user, even if the speedups are not the best ones that could be obtained by manually optimized implementations. We plan to remove the limits explained above. Other perspectives include the following points.

Shared memory. The ADP compiler could be improved to better use the memory hierarchy of the card. In the Nvidia architecture, a 16 KB *shared* memory is available for the threads in the same block. This local memory is very fast and should be used to maximize the efficiency, for example in storing portions of some dynamic programming tables. This memory is not used in our current implementation. Of course, the best usage of the shared memory depends on the application: for now, we did not find a generic way to determine from the grammar this best usage. Some hints given in the grammar file could indicate to the compiler which dynamic programming tables should be handled in this way.

Static evaluation of grammars. We plan to test other grammars, in bioinformatics as in other domains. Which grammars are efficient to parallelize, and why?

Other targets. We plan to test the ADP methodology on other manycore architectures, in particular through the new OpenCL standard [1]. Again, the fact that the ADP methodology is generic allows to write portable solutions.

Acknowledgements

Part of this research was done during P. Steffen's stay in Université Lille 1. This research was carried through the "NVIDIA Professor Partnership" program.

References

1. The Khronos Group, OpenCL 1.0 specification, 2008.
2. Nvidia CUDA programming guide 2.0, 2008.
3. A.V. Aho and J.D. Ullman. *The Theory of Parsing, Translation and Compiling.* Prentice-Hall, Englewood Cliffs, NJ, 1973. I and II.
4. Maria Charalambous, Pedro Trancoso, and Alexandros Stamatakis. Initial experiences porting a bioinformatics application to a graphics processor. *Adv. in Informatics*, pages 415–425, 2005.
5. R. Durbin, S. Eddy, A. Krogh, and G. Mitchison. *Biological Sequence Analysis.* Cambridge University Press, 1998.

6. R. Giegerich, C. Meyer, and P. Steffen. A discipline of dynamic programming over sequence data. *Science of Computer Programming*, 51(3):215–263, 2004.
7. R. Giegerich and P. Steffen. Challenges in the compilation of a domain specific language for dynamic programming. In *Proceedings of the 2006 ACM Symposium on Applied Computing*, 2006.
8. Mathieu Giraud and Jean-Stéphane Varré. Parallel position weight matrices algorithms. In *International Symposium on Parallel and Distributed Computing (ISPDC 2009)*, 2009.
9. Lukasz Ligowski and Witold Rudnicki. An efficient implementation Smith-Waterman algorithm on GPU using CUDA, for massively parallel scanning of sequence databases. In *IEEE International Workshop on High Performance Computational Biology (HiCOMB 2009)*, 2009.
10. Weiguo Liu, Bertil Schmidt, Gerrit Voss, and Wolfgang Müller-Wittig. GPU-ClustalW: using graphics hardware to accelerate multiple sequence alignment. In *High Performance Computing (HiPC 2006), LNCS 4297*, pages 363–374, 2006.
11. Yongchao Liu, Douglas Maskell, and Bertil Schmidt. CUDASW++: optimizing Smith-Waterman sequence database searches for CUDA-enabled graphics processing units. *BMC Research Notes*, 2(1):73, 2009.
12. Yongchao Liu, Bertil Schmidt, and Douglas Maskell. Parallel reconstruction of neighbor-joining trees for large multiple sequence alignments using CUDA. In *IEEE International Workshop on High Performance Computational Biology (HiCOMB 2009)*, 2009.
13. Svetlin A Manavski and Giorgio Valle. CUDA compatible GPU cards as efficient hardware accelerators for Smith-Waterman sequence alignment. *BMC Bioinformatics*, 9 Suppl 2:S10, 2008.
14. D. Mathews, J. Sabina, M. Zuker, and D. Turner. Expanded sequence dependence of thermodynamic parameters improves prediction of RNA secondary structure. *Journal of Molecular Biology*, (288):911–940, 1999.
15. R. Nussinov, G. Pieczenik, J.R. Griggs, and D.J. Kleitman. Algorithms for loop matchings. *SIAM J. Appl. Math.*, 35:68–82, 1978.
16. J. Reeder and R. Giegerich. Design, implementation and evaluation of a practical pseudoknot folding algorithm based on thermodynamics. *BMC Bioinformatics*, 5(104), 2004.
17. J. Reeder, J. Reeder, and R. Giegerich. Locomotif: From graphical motif description to RNA motif search. *Bioinformatics*, 23(13):391–400, 2007.
18. Guillaume Rizk and Dominique Lavenier. GPU accelerated RNA folding algorithm. In *Using Emerging Parallel Architectures for Computational Science / International Conference on Computational Science (ICCS 2009)*, 2009.
19. Michael C Schatz, Cole Trapnell, Arthur L Delcher, and Amitabh Varshney. High-throughput sequence alignment using graphics processing units. *BMC Bioinformatics*, 8:474, 2007.
20. D.B. Searls. Linguistic approaches to biological sequences. *CABIOS*, 13(4):333–344, 1997.
21. Haixiang Shi, Bertil Schmidt, Weiguo Liu, and Wolfgang Mueller-Wittig. Accelerating error correction in high-throughput short-read DNA sequencing data with CUDA. In *IEEE International Workshop on High Performance Computational Biology (HiCOMB 2009)*, 2009.
22. P. Steffen. *Compiling a Domain Specific Language for Dynamic Programming.* PhD thesis, Bielefeld University, 2006.
23. P. Steffen and R. Giegerich. Versatile and declarative dynamic programming using pair algebras. *BMC Bioinformatics*, 6(224), 2005.